Using Large Language Models (LLMs) for Test Generation

Tejal Devshetwar, Niharika Pathuri, Raghuram Guddati, Varun Yeduru

May 6, 2025

Introduction / Motivation:

Testing is an important aspect of writing code. We need to verify if the code written is correct or not. To do so, developers write test cases for the written code. In software development, writing unit tests is often one of the most difficult and ignored tasks. We are taught the importance of testing, but writing good test cases takes a lot of time and effort. Having a great coverage of all the edge cases can be exhausting when done manually. This process can be sped up by using LLMs. There are many scenarios where Large Language Models (LLMs) do not provide correct test generations. We are interested in exploring how effectively they can automate this process.

To help us achieve this task, we are trying to use TestPilot 2, a tool that automatically generates unit tests using an LLM. Instead of manually figuring out expected outputs, the tool analyzes functions in JavaScript/TypeScript and asks the LLM to write tests. This promises to save time, reduce human error, and improve code coverage.

There are many scenarios where Large Language Models (LLMs) do not provide correct test generations. We are interested in exploring how effectively they can automate this process [Zhang et al., 2024].

Previous / Existing Approaches:

Before the use of LLMs for test generation, several established techniques were commonly used in software testing. Some of the tools and approaches we learned in this course include:

- EvoSuite: Uses algorithms to automatically generate JUnit tests that maximize code coverage and detect faults present in them.
- Combinatorial Interaction Testing (CIT): Systematically covers combinations of input parameters to detect interaction faults.
- JaCoCo: A code coverage analysis tool for Java that is used to measure how much of the code is exercised by tests.It generates reports that visually show the developers the code coverage levels.

- Fuzzing: Generates random or unexpected input to uncover bugs, vulnerabilities, or edge cases in software. To implement fuzzing, tools are usually used.
- GUI Testing: Automates interaction with graphical user interfaces to check for correct behavior. In this we check each element of our interface and verify if it performs the desired task or not.
- Test Specification Languages (TSL): Provides a formal way to describe test cases, expected inputs, and outputs. This is a structured way of writing tests from requirements.

These tools often require manual setup or domain expertise. LLM-based tools like Test-Pilot 2 offer an alternative by automatically generating human-readable test cases with minimal manual effort. It is not necessary that these approaches understand high-level code intent. This is one of the other gaps that LLMs aim to address.

Technical Description:

TestPilot 2 is a tool that uses LLMs, such as GPT, to automatically generate unit tests for JavaScript/TypeScript programs. Instead of relying on human developers, TestPilot 2 analyzes the codebase, identifies public functions, and prompts the LLM to write tests. In human-written test cases the room for error is huge and achieving high coverage requires substantial resources. Automating this process can dramatically improve productivity.

Recent research also demonstrates that integrating static analysis with LLMs can further improve the quality and coverage of automatically generated tests, and enable multi-language support, as shown in the ASTER pipeline for Java and Python (Pan et al., 2024)[3][4]

The overall workflow is illustrated in the following diagram:

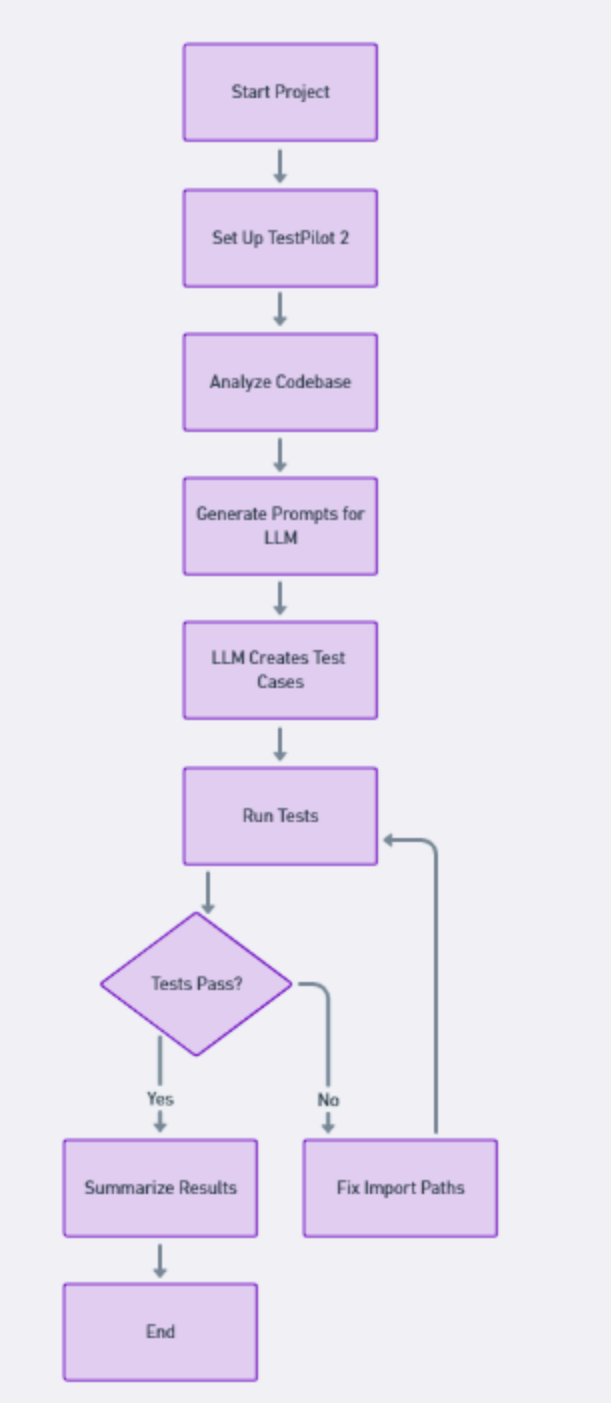


Figure 1: Workflow of Test Generation Using TestPilot 2

The flowchart shows the basic steps TestPilot 2 follows to create unit tests automatically:

- Start Project

You begin by setting up a new project where you want to generate tests.

- Set Up TestPilot 2
  You install and configure TestPilot 2, including API keys and necessary files.

- Analyze Codebase
  The tool scans your code to find functions that need testing.

- Generate Prompts for LLM
  It creates prompts to ask the language model, like “Write a test for this function.”

- LLM Creates Test Cases
  The language model (like GPT-3.5) writes unit tests using JavaScript frameworks such as Mocha.

- Run Tests
  The tests that were generated are executed to make sure they are right.

  If the tests pass, we move on to collect test coverage, which is displayed directly in the command line.

  If they fail, it's usually because some import paths or required files are incorrect.

- Fix Import Paths
  A custom script fixes broken import paths so tests can run correctly.

- Summarize Results
  You evaluate how well it worked: time saved, bugs found, and code coverage improved.

TestPilot 2 operates in two major stages:

1) Prompt Generation

TestPilot 2 automatically crafts prompts to send to the LLM. For example, it might generate a prompt like, “Write a Jest unit test for the deleteUser function that tests valid input and error cases.” Specific to our example, it generated a prompt like the one shown below:

````
Your task is to write a test for the following function:
```

math-utils.add(a, b)
```




Please proceed by modifying the following code fragment:
```
let mocha = require('mocha');
let assert = require('assert');
let math_utils = require('math-utils');
describe('test math_utils', function() {
    it('test math-utils.add', function(done) {

```
so that it becomes a test suite containing a few self-contained unit tests.  The tests should not rely on any
external resources. For example, a test should not attempt to access files that it does not create itself.

Provide your answer as a fenced code block.
````

2) Test Generation and Output

The LLM returns generated test cases, typically using a framework like Jest or Mocha for JavaScript. TestPilot 2 uses Mocha to return generated test cases. The tool integrates these tests into the codebase, where they can be reviewed and refined by developers.

Evaluation

We evaluated TestPilot 2 by creating a small project called math-utils. In this project, we wrote functions like add, factorial, and isPrime in a file named index.js. These functions include basic operations such as adding two numbers, calculating factorial using recursion, and checking if a number is prime. The code we used is shown below.

```
function add(a, b) {
  return a + b;
}

function factorial(n) {
  if (n < 0) throw new Error("Negative input not allowed");
  return n === 0 ? 1 : n * factorial(n - 1);
}

function isPrime(num) {
  if (num <= 1) return false;
  for (let i = 2; i <= Math.sqrt(num); i++) {
    if (num % i === 0) return false;
  }
  return true;
}

module.exports = {
  add,
  factorial,
  isPrime
};
```

Setup

- We started by creating a new project and cloning the TestPilot 2 repository into it, as shown in Figure 2.

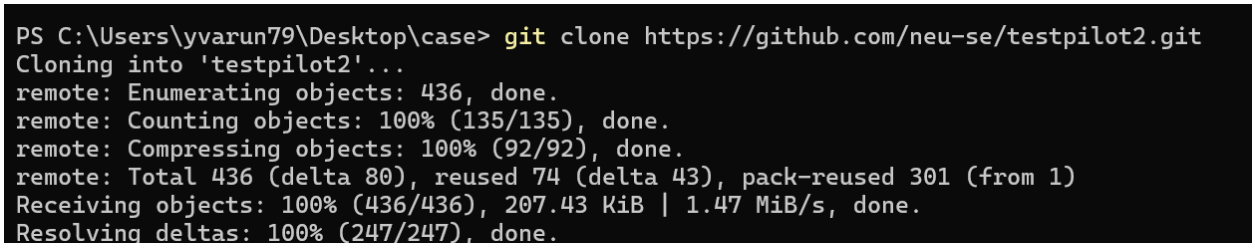
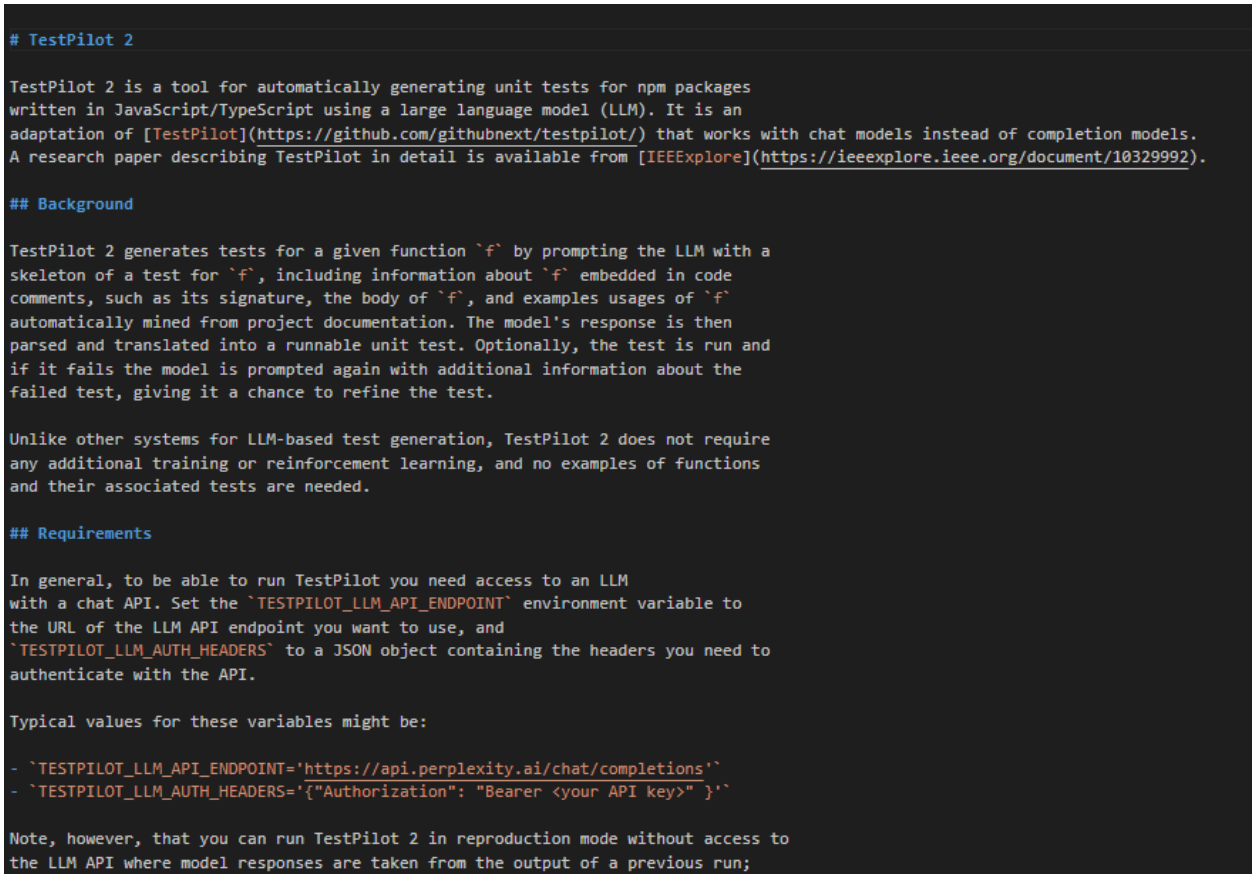


Figure 2: Cloning the TestPilot 2 Repository from GitHub Using the git clone Command

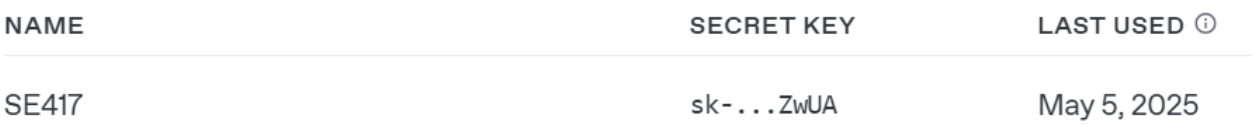
- Then we installed dependencies as instructed in the GitHub README file, a portion of which is shown in Figure 3.



```
# TestPilot 2

TestPilot 2 is a tool for automatically generating unit tests for npm packages
written in JavaScript/TypeScript using a large language model (LLM). It is an
adaptation of [TestPilot](https://github.com/githubnext/testpilot/) that works with chat models instead of completion models.
A research paper describing TestPilot in detail is available from [IEEExplore](https://ieeexplore.ieee.org/document/10329992).

## Background

TestPilot 2 generates tests for a given function `f` by prompting the LLM with a
skeleton of a test for `f`, including information about `f` embedded in code
comments, such as its signature, the body of `f`, and examples usages of `f`
automatically mined from project documentation. The model's response is then
parsed and translated into a runnable unit test. Optionally, the test is run and
if it fails the model is prompted again with additional information about the
failed test, giving it a chance to refine the test.

Unlike other systems for LLM-based test generation, TestPilot 2 does not require
any additional training or reinforcement learning, and no examples of functions
and their associated tests are needed.

## Requirements

In general, to be able to run TestPilot you need access to an LLM
with a chat API. Set the `TESTPILOT_LLM_API_ENDPOINT` environment variable to
the URL of the LLM API endpoint you want to use, and
`TESTPILOT_LLM_AUTH_HEADERS` to a JSON object containing the headers you need to
authenticate with the API.

Typical values for these variables might be:

- `TESTPILOT_LLM_API_ENDPOINT='https://api.perplexity.ai/chat/completions'`
- `TESTPILOT_LLM_AUTH_HEADERS='{"Authorization": "Bearer <your API key>" }'`

Note, however, that you can run TestPilot 2 in reproduction mode without access to
the LLM API where model responses are taken from the output of a previous run;
```

Figure 3: TestPilot 2 Repository's README File Showing Setup Instructions

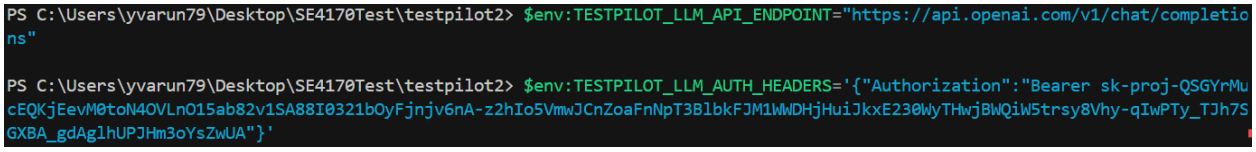
- The next step was to create an OpenAI API key. For our project, we decided to use GPT-3.5, as shown in Figure 4.



NAME	SECRET KEY	LAST USED ⓘ
SE417	sk-...ZwUA	May 5, 2025

Figure 4: Generating an OpenAI API Key for Use with TestPilot 2

- Upon creating the key, we had to set the TESTPILOT_LLM_API_ENDPOINT to the URL of the LLM API endpoint and TESTPILOT_LLM_AUTH_HEADERS to a JSON object of headers to authenticate the API, as shown in Figure 5. (LLM4SoftwareTesting, 2025)



```
PS C:\Users\yvarun79\Desktop\SE4170Test\testpilot2> $env:TESTPILOT_LLM_API_ENDPOINT="https://api.openai.com/v1/chat/completio
ns"

PS C:\Users\yvarun79\Desktop\SE4170Test\testpilot2> $env:TESTPILOT_LLM_AUTH_HEADERS='{"Authorization":"Bearer sk-proj-QSGYrMu
cEQKjEevM0toN4OVLnO15ab82v1SA88I0321bOyFjnjv6nA-z2hIo5VmwJCnZoaFnNpT3BlbkFJM1WWDHjHuiJkxE230WyTHwjBWQiW5trsy8Vhy-qIwPTy_TJh7S
GXBA_gdAglhUPJHm3oYsZwUA"}'
```

Figure 5: Setting Environment Variables for OpenAI API Endpoint and Authentication

- We built and ran the tool on example JavaScript/TypeScript files, as shown in Figure 6.

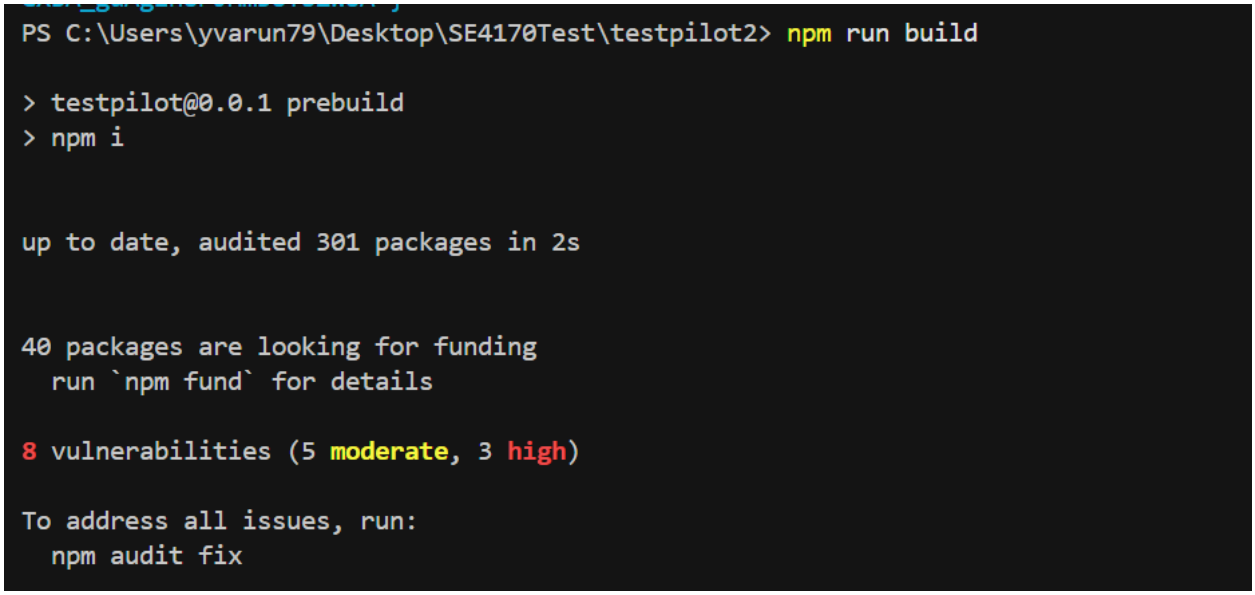


Figure 6: Running the Build Command for TestPilot 2 Using NPM

- We also had to modify some code files to make the tool behave the way we want for our program. Mainly, we updated mochavalidator.ts in both the src and dist directories.
- Also, we created a new script file fix-import.js to automatically correct import paths in the generated test cases, which were initially pointing to incorrect locations, as shown in Figure 7.

```
const fs = require('fs');
const path = require('path');

const testDir = path.join(__dirname, '../math-utils');

// Find all subdirectories starting with 'test-'
fs.readdirSync(testDir).forEach((subdir) => {
  const subdirPath = path.join(testDir, subdir);

  if (fs.statSync(subdirPath).isDirectory() && subdir.startsWith('test-')) {
    const files = fs.readdirSync(subdirPath).filter((f) => f.endsWith('.js'));

    files.forEach((file) => {
      const filePath = path.join(subdirPath, file);
      let content = fs.readFileSync(filePath, 'utf8');

      // Replace any kind of require('./math-utils') or require('../math-utils') with require('..')
      content = content.replace(
        /require\((['"`])(\.\/|\.\.\/)*math-utils\1\)/g,
        "require('..')"
      );

      fs.writeFileSync(filePath, content, 'utf8');
      console.log(`Fixed imports in ${filePath}`);
    });
  }
});
```

Figure 7: Script fix-imports.js Used to Correct Broken Import Paths in Test Files

Observations

The main command used to run the test generation process is shown in Figure 8. It includes parameters such as the model to be used (gpt-3.5-turbo), the template files, the target package (../math-utils), and other configuration flags like snippet length and retry behavior.

Figure 8: Command to Generate Test Cases Using TestPilot 2 with GPT-3.5 Turbo

As a result, some test cases failed due to module resolution errors or missing files.

For example, a few generated test files failed to run because the test runner could not locate the required modules, leading to errors like ENOENT: no such file or directory for coverage reports. Other tests executed without issues, indicating that the generation and execution pipeline partially worked as intended.

To address the broken imports:

- We created a custom script called fix-import.js to automatically detect and correct incorrect import paths in the generated test files, as shown in Figure 9.

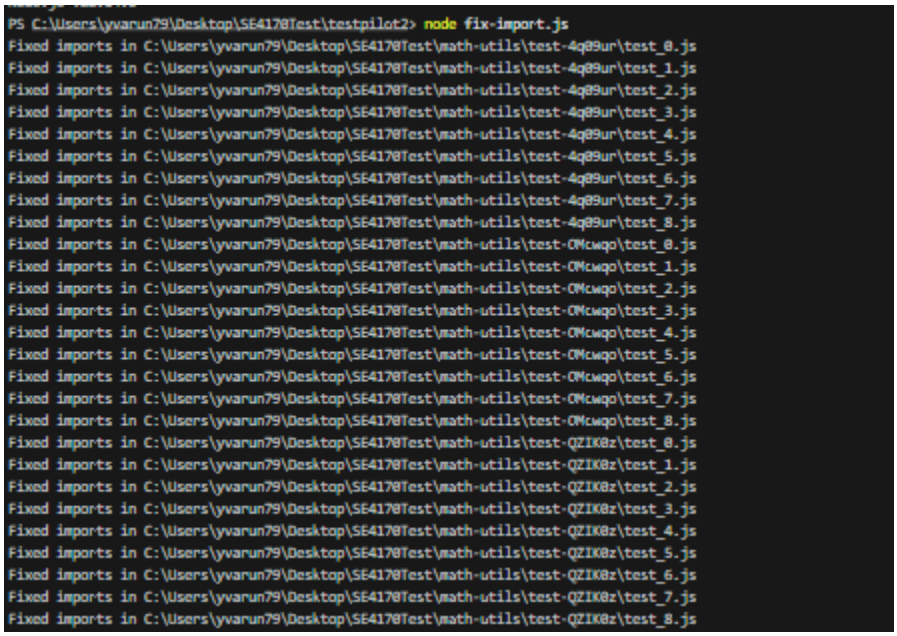


Figure 9: Executing fix-import.js to Correct Import Paths in Generated Test Files

- After running this script, we used the command npx nyc --reporter=text mocha "test-*/test_*.js" to execute the updated tests and collect coverage data, as shown in Figure 10.

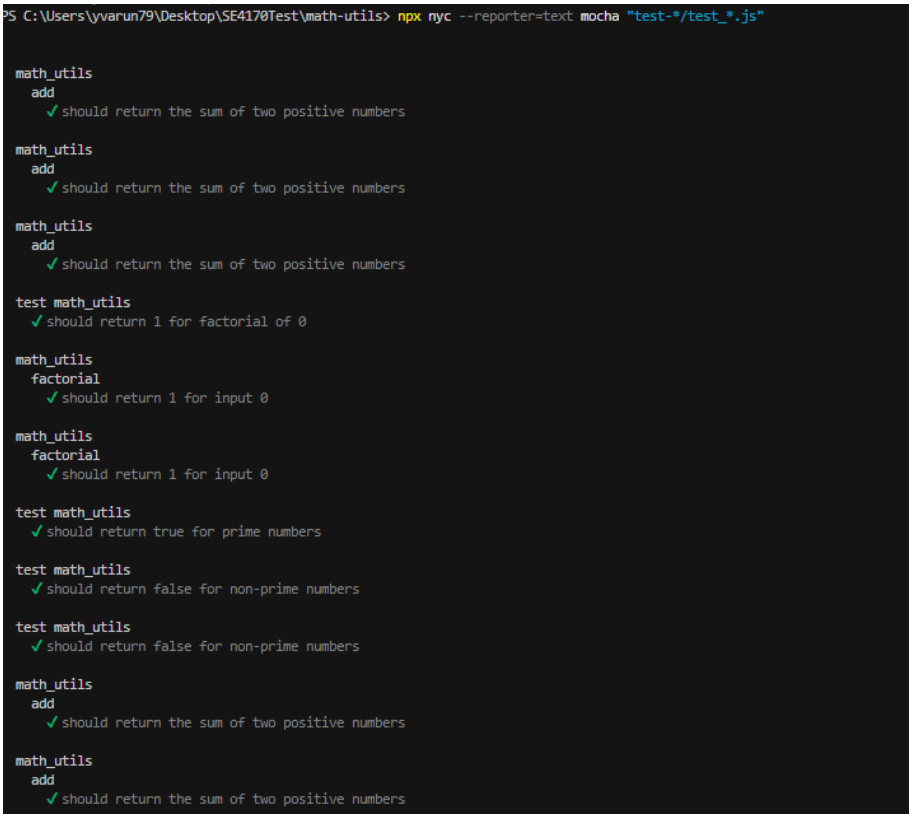
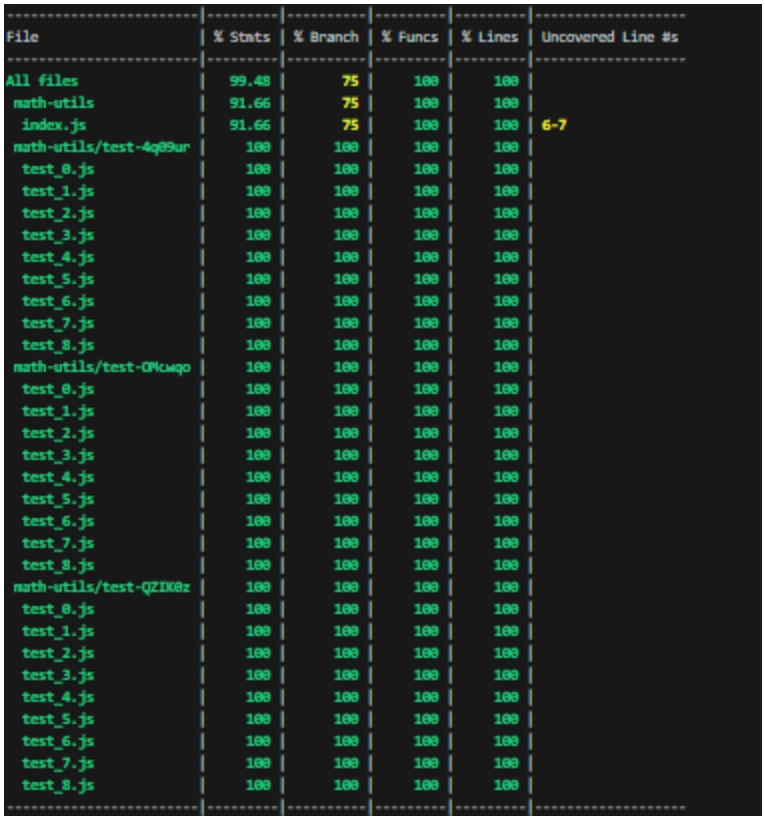


Figure 10: Running Generated Test Cases Using Mocha



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	99.48	75	100	100	
math-utils	91.66	75	100	100	
index.js	91.66	75	100	100	6-7
math-utils/test-4q09ur	100	100	100	100	
test_0.js	100	100	100	100	
test_1.js	100	100	100	100	
test_2.js	100	100	100	100	
test_3.js	100	100	100	100	
test_4.js	100	100	100	100	
test_5.js	100	100	100	100	
test_6.js	100	100	100	100	
test_7.js	100	100	100	100	
test_8.js	100	100	100	100	
math-utils/test-OMcwqo	100	100	100	100	
test_0.js	100	100	100	100	
test_1.js	100	100	100	100	
test_2.js	100	100	100	100	
test_3.js	100	100	100	100	
test_4.js	100	100	100	100	
test_5.js	100	100	100	100	
test_6.js	100	100	100	100	
test_7.js	100	100	100	100	
test_8.js	100	100	100	100	
math-utils/test-QZIX0z	100	100	100	100	
test_0.js	100	100	100	100	
test_1.js	100	100	100	100	
test_2.js	100	100	100	100	
test_3.js	100	100	100	100	
test_4.js	100	100	100	100	
test_5.js	100	100	100	100	
test_6.js	100	100	100	100	
test_7.js	100	100	100	100	
test_8.js	100	100	100	100	

Figure 11: Code Coverage Report

Figure 11 shows the code coverage report generated after executing the test cases. With most of the test files attaining 100% coverage, the automatically produced tests effectively exercised all code components. One exception is in index.js, which displays uncovered lines at 6-7 and 75% branch coverage. The overall performance of the LLM-generated tests in Figure 11 shows how very successful they were at covering the codebase.

Advantages

- Saved time by automatically generating unit tests instead of writing them manually.
- Reduced mistakes like syntax errors or missing edge cases in test code.
- Improved code coverage as most files reached 100 percent as shown in Figure 11.
- Generated tests were clear, easy to read, and followed a consistent structure.

Disadvantages

- Some generated tests were too simple or repeated similar checks without adding useful coverage.
- Setting up TestPilot 2 was challenging at first and required extra time to configure environment variables and dependencies.
- The tool struggled with complex functions that needed a deeper understanding of the business logic.
- We had to manually fix broken import paths using a custom script before the tests could run correctly.

Performance Metrics

For details on how the tests ran and the issues we encountered, see the Observations section.

In summary:

- Some test files needed manual fixes (like import paths).
- We used fix-import.js and re-ran tests with npx nyc to get coverage.
- After adjustments, we observed improved code coverage and test stability.

Summary and Recommendation

With TestPilot2, we had the opportunity to investigate an LLM-based solution for automating unit test creation. For generating basic tests for utility functions with simple input and output behavior, we found it helpful. It saved time and reduced physical effort, thereby improving the development process's functionality at times.

TestPilot2 has obvious restrictions, though, as compared to programs like EvoSuite and JaCoCo, which we examined in class. While JaCoCo gave far better information on which sections of the code were tested, EvoSuite was more successful in creating thorough JUnit tests with high code coverage. TestPilot 2 frequently generated too simplistic or repetitious test cases and battled more complicated reasoning. To get it operating, we also had to spend time defining environment variables and repairing problems including broken import paths.

Recommendations:

- We suggest using TestPilot2 early in development to get quick test coverage for straightforward functions

- For more advanced or critical testing, tools like EvoSuite and JaCoCo are more reliable and produce better results

- It's important to manually review all LLM-generated tests to make sure they are actually meaningful and correct.

- LLM tools are helpful for reducing routine work, but they still need human guidance to ensure quality and completeness

Overall, TestPilot2 is a useful tool for generating a starting point, but it doesn't replace the need for strong test generation and coverage tools like the ones we've already learned in this course.

References

1) GitHub Next, “TestPilot,” GitHub repository. https://github.com/githubnext/testpilot
2) ChatDev: Collaborative Software Development with Chatbots. https://ieeexplore.ieee.org/document/10329992
3) LLM4SoftwareTesting, GitHub repository. https://github.com/LLM-Testing/LLM4SoftwareTest
4) Zhang, Q., Shang, Y., Fang, C., Gu, S., Zhou, J., & Chen, Z. (2024). TestBench: Evaluating Class-Level Test Case Generation Capability of Large Language Models. arXiv preprint arXiv:2409.17561.
5) Pan, R., Kim, M., Krishna, R., Pavuluri, R., & Sinha, S. (2024). ASTER: Natural and Multi-language Unit Test Generation with LLMs. arXiv preprint arXiv:2409.03093.